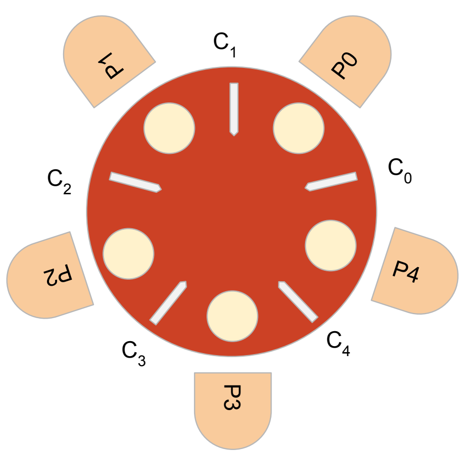
Deadlock

Tasks Using Multiple Resources





Dining (Chinese) Philosophers



- N (Chinese) philosophers are either thinking or dining at a round table
  - P0, P1, P2, ….
- They share chopsticks with their neighbors
  - Chopsticks: C0, C1, C2, …
- Each philosopher needs TWO chopsticks to start dining
- P_k requires “left” chopstick C_k and “right” chopstick C_{k+1}

Dining Philosophers (First Attempt)

```
/* Philosopher-k */

while (true) {

  chop[k].wait();     // left chopstick
  chop[k+1].wait();   // right chopstick, (k+1) mod 5

  /* DINE */

  chop[k].signal();
  chop[k+1].signal();

  /* THINK */
}
```

I.P.

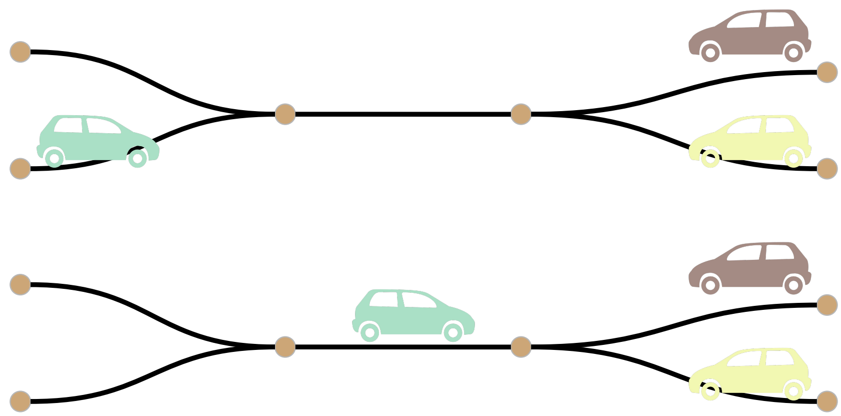
What could go wrong with the above algorithm?

- Interruptible points
- Each philosopher holding a left chopstick, while waiting for the right chopstick

DEADLOCK

Deadlock Formalism

Road Deadlock (“Process” & “Resource”)



Deadlock

- Two or more processes (or threads) are in a deadlock state when they are waiting for an **event** that can only be triggered by these processes alone
- Our focus
  - A finite number of **resources (M)** to be distributed among **N competing processes**
- Resource Use pattern
  - Each process follows this sequence: **request - use - release**

Necessary vs. Sufficient Condition

A ⇒ B
sufficient   necessary

A ⇒ B

- Reads "(if) A (then) B"
  - A is the **sufficient** condition for B
  - B is the **necessary** condition for A
- We **can't** conclude B ⇒ A
- But we *can infer* its ***contrapositive*** **not** B ⇒ **not** A

"**Live in MI**" ⇒ "**Latitude ≥ 41° N**"

- (if) **you live in MI** (then) your **latitude is at least** 41°
- We ***can't conclude*** (if) "**latitude is at least 41°**" then "**you live in MI**"
- However, "if your latitude is **less than** 41°" then "you DON'T live in MI"

Logical Opposite

Which one is the logical opposite of "A and B"?

1. not A and not B
2. not A or not B
3. not (A or B)

Notations:

- A and B   A ∧ B
- P or Q   P ∨ Q
- not S   ¬S

Necessary Conditions for Deadlock

Resource

Process

When a system is in a deadlock state, then **ALL** of the following (*necessary*) conditions must be true

- **Mutual Exclusion** (ME): resources are held in a non-sharable mode
- **No-Preemption** (NP): resources cannot be preempted from its current holder
- **Hold & Wait** (HW): a process must be holding (at least) a resource while waiting to acquire more resources
- **Circular Wait** (CW): there is a set of processes P0, P1, ..., Pn such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., and Pn is waiting for a resource held by P0

$$DL \Rightarrow ME \text{ and } NP \text{ and } HW \text{ and } CW$$

DL ⇒ ME **and** HW **and** NP **and** CW

Fallacy: ~~$ME \land HW \land NP \land CW \Rightarrow DL$~~

ContraPos: ¬ME **or** ¬HW **or** ¬NP **or** ¬CW ⇒ ¬DL

Deadlock Prevention Mechanisms

- *Deny* **Mutual Exclusion**
  - Do not allow process to use resources exclusively, force them to always share
- *Deny* **No-Preemption** **(***Allow* **Preemption)**
  - Allow resources to be preempted/"stolen" from their current holder
- *Deny* **Hold-and-Wait** **(***Allow* **Hold-only or** *Allow* **Wait-Only)**
  - Enforce **all-or-nothing** policy, a process that requires multiple resources must acquire them all at the same time
- *Deny* **Circular Wait**
  - Enforce **resource-ordering** policy. Assign resources into different "rings/levels". Resource must be requested in increasing order of these rings/levels

Resource Ordering Policy

1. Class 0: USB Stick and RAM
2. Class 1: Printer
3. Class 2: GPU

Policy: processes must request the resources in *increasing* class number

Example: Process X needs both USB stick and GPU

1. Allow: request(USB), request(GPU)
2. Deny: request(GPU), request(USB)

Resource Ordering Policy

1. Class 0: USB Stick and RAM
2. Class 1: Printer
3. Class 2: GPU

Policy: Resources in *same* class number must be requested together

Example: Process Y needs **both** USB stick and RAM

1. Deny: request(USR), request(RAM)
2. Deny: request(RAM), request(USB)
3. Allow: request(USB, RAM)

Dining Philosophers Deadlock Prevention

- Allow at most N-1 philosophers to dine simultaneously
- Apply <u>asymmetric</u> chopstick pick up order
  - Left-Handed philosophers: pick left chopstick first
  - Right-Handed Philosophers: pick right chopstick first
- Other strategies?

Dining Philosophers: Limit N-1 diners

```
/* (prone to deadlock) */

while (true) {

  chop[k].wait();    // left chopstick
  chop[k+1].wait();  // right chopstick

  /* DINE */

  chop[k].signal();
  chop[k+1].signal();

  /* think */
}
```

```
/* Philosopher-k (deadlock free) */

while (true) {
  diner.wait();
  chop[k].wait();    // left chopstick
  chop[k+1].wait();  // right chopstick

  /* DINE */

  chop[k].signal();
  chop[k+1].signal();
  diner.signal();
  /* think */
}
```

```
Semaphore chop[5] = {1, 1, 1, 1, 1};
Semaphore diner = 4;
```

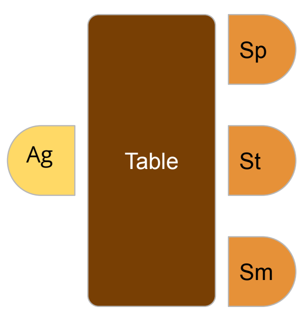
Cigarette Smokers Problem

- Four concurrent threads: one agent and three smokers
- Each smoker needs three ingredients: paper, tobacco, and a match
- The agent has infinite amount of ALL the ingredients
- Each smoker has infinite amount of ONLY ONE ingredient
- The agent randomly select two ingredients and make them available to the smokers

Cigarette Smokers (unsynchronized code)

Sp: smoker with paper (needs tobacco and match)
St: smoker with tobacco (needs paper and match)
Sm: smoker with match (need paper and tobacco)

```
// agent
while (1) {

  /* two distinct items */
  (t1,t2) = rnd_two_items();

  place_supplies (t1, t2);

}
```

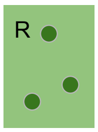



```
// smoker-X (needs Y and Z)
while (1) {

  y = get_supply();

  z = get_supply();

  smoke();

}
```

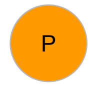
the table can hold ONLY TWO items max

Resource Allocation Graph

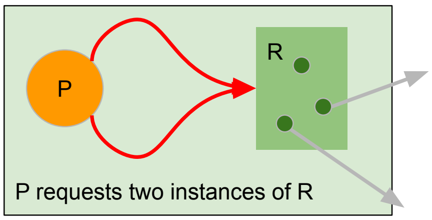
- Nodes: processes or resources
- Edges: request edges / assignment edges



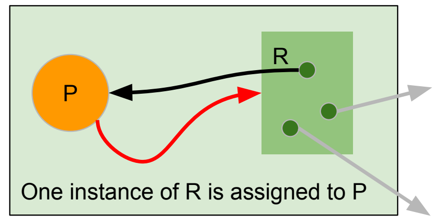
3 instances of resource R



process P

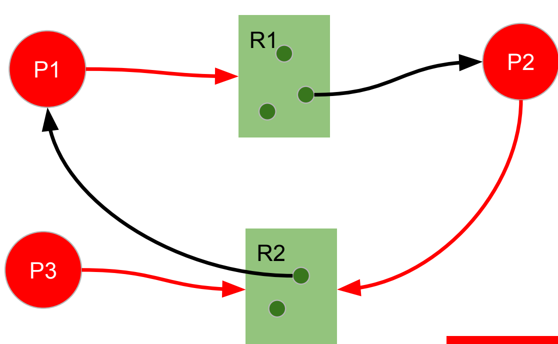


REQUEST EDGE



ASSIGNMENT/ALLOCATION EDGE

Resource Allocation Graph



P1 (**blocked**): is holding one instance of R2, and is waiting for one instance of R1 (*can be granted immediately*)

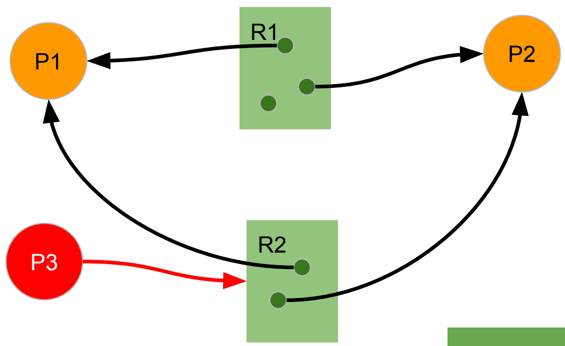
P2 (**blocked**): is holding one instance of R1 and is waiting for one instance of R2

P3 (**blocked**): is waiting for one instance of R2

P2 and P3 compete for R2

Can't conclude: Cycle ⇒ Deadlock
Can't conclude CW ⇒ DL

Resource Allocation Graph



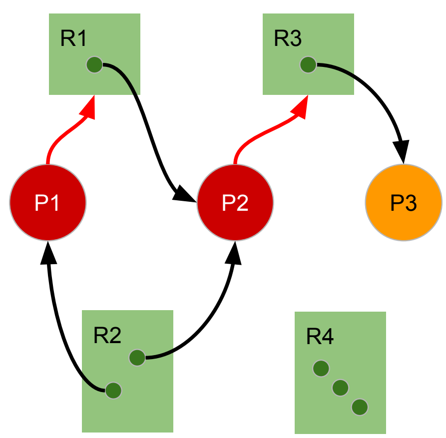
P1 (**ready/running**): is holding one instance of R2, and **one instance of R1**

P2 (**ready/running**): is holding one instance of R1 and is waiting for one instance of R2

P3 (**blocked**): is waiting for one instance of R2 (**can't be immediately granted**)

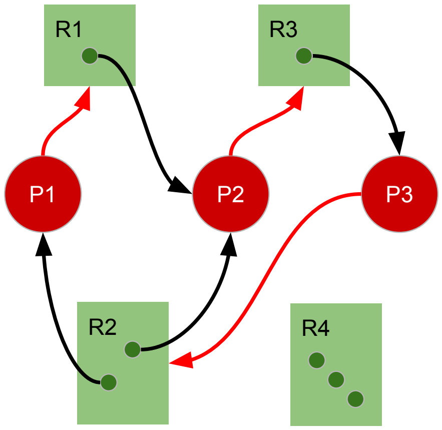
Can conclude: No CW ⇒ No DL

Resource Allocation Graph



P1 (blocked)
P2 (blocked)
P3 (ready/running)

Resource Allocation Graph



P1 (blocked)
P2 (blocked)
P3 (blocked)